Communication Studies of DMP and SMP Machines

Andrew Sohn
Dept. of Computer and Information Science
New Jersey Institute of Technology, Newark, NJ 07102-1982
sohn@cis.njit.edu

Rupak Biswas
MRJ Technology Solutions, Mail Stop T27A-1
NASA Ames Research Center, Moffett Field, CA 94035-1000
rbiswas@nas.nasa.gov

Abstract

Understanding the interplay between machines and problems is key to obtaining high performance on parallel machines. This paper investigates the interplay between programming paradigms and communication capabilities of parallel machines. In particular, we explicate the communication capabilities of the IBM SP-2 distributed-memory multiprocessor and the SGI PowerCHALLENGEarray symmetric multiprocessor. Two benchmark problems of bitonic sorting and Fast Fourier Transform are selected for experiments. Communication-efficient algorithms are developed to exploit the overlapping capabilities of the machines. Programs are written in Message-Passing Interface for portability and identical codes are used for both machines. Various data sizes and message sizes are used to test the machines' communication capabilities. Experimental results indicate that the communication performance of the multiprocessors are consistent with the size of messages. The SP-2 is sensitive to message size but yields a much higher communication overlapping because of the communication co-processor. The PowerCHALLENGEarray is not highly sensitive to message size and yields a low communication overlapping. Bitonic sorting yields lower performance compared to FFT due to a smaller computation-to-communication ratio.

1 Introduction

Programming parallel machines involves numerous practical concerns. The problems under consideration need to be carefully studied to identify potential parallelism. Suitable algorithms for the problems need to be developed to manifest their potential parallelism. Programming must be done in a way to effectively realize the algorithms and harness their parallelism for the target machines. Each machine's special capabilities, if any, should be utilized to improve the performance. The programmer must have a good understanding of the characteristics of the problem as well as of the underlying machine architecture. Programmability that refers to the easiness of programming parallel machines is one of the keys to success. It is particularly important for problems requiring complex synchronization and parallelization.

Functional languages have been used to overcome the difficulties in programming multiprocessors. They are different from programs written in imperative languages in that they do not require explicit parallel constructs such as message-passing or synchronization primitives. The programmer need not understand the low level implementation details of the machine architecture, data distribution, synchronization, and so on. Instead, one can concentrate on algorithmic improvement or the quality of solutions. The non-strict functional language Id [16] has been ported to the Monsoon data-flow multiprocessor [18], and demonstrated that functional languages can be an alternative to parallel programming. The strict functional language Sisal [14] and its optimizing compiler OSC [5] demonstrated that functional languages can yield high performance on shared-memory machines [7].

However, the performance of functional languages on distributed-memory machines is yet to be determined. High Performance Fortran (HPF) aims at providing programmability for distributed-memory multiprocessors [9]. Data parallelism in large arrays are a typical target parallelism used in HPF [11,12]. A simple description of data distribution such as blocked or cyclic can help keep programmers away from the low-level details of data distribution and communication issues. Programmers instead concentrate on the algorithmic issues of the problem under consideration. Analyzing the behavior of the program, data can be partitioned and allocated to processors such that runtime data movement can be minimized.

One of the key issues behind these approaches are to minimize communication and synchronization times. Large-scale machines distribute data in a way that there is no overlapping or copying of major data. Typical distributed-memory machines incur a lot of latency, ranging from a few μs to tens of μs for a single remote read operation. The SP-2, for example, requires about 40 μs to read data allocated to remote processors. Considering that the microprocessors are running at 66.7 MHz (15 ns cycle time) for the SP-2 590 model, the loss due to a single remote read operation is enormous: more than 2600 cycles. Several models developed to capture the communication behavior include BSP [22], LogP [6], and LogGP [2]. The LogP model defines four parameters: L for latency, o for overhead, g for gap between messages, and P for processors. Each of the four parameters is designed to capture a certain aspect of communication behavior for short messages. The LogGP model extends the LogP model by adding G to capture the communication behavior of long messages, where G is the gap between bytes of the same message.

Multithreading aims at tolerating remote memory latency through split-phase read mechanism and context switching [19,8,10,13,17]. Threads are usually delimited by remote read instructions which may incur long latency if the requested data is located in a remote processor [17]. Through a split-phase read mechanism, a processor switches to another thread instead of waiting for the requested data to arrive, thereby masking the detrimental effect of latency. The Monsoon dataflow machine switches context every instruction, where a thread consists of a single instruction [18]. The Tera multithreaded architecture (MTA) provides hardware support for multithreading [3]. The maximum of 128 threads are provided per processor. Context switch takes place whenever a remote load or synchronizing load is en-

countered. Experimental results indicated that multithreading can help lessen the impact caused by the mismatch of data distribution to workload distribution [20].

In this paper, we investigate the interplay between programming paradigms and communication capabilities of parallel machines. In particular, we explicate the communication capabilities of the IBM SP-2 distributed-memory multiprocessor [1,21] and the SGI Power CHALLENGEarray symmetric multiprocessor. We fix the programming paradigm to Message Passing Interface (MPI) [15] to show programmability and to identify how performance changes at the expense of programmability. We first give a brief description of the two machines in Section 2. Section 3 explains how two benchmark problems (bitonic sorting and fast fourier transform) are modified to fit the requirements of our communication studies. Section 4 gives an overview of the experimental results and presents several key observations. Sections 5 to 7 elaborate on the communication efficiency across problems, machines, message sizes, and number of processors.

2 Two Multiprocessor Machines

2.1 IBM SP-2

The SP-2 is developed at IBM Scalable Power Parallel Systems [1,21]. Processors are POWER2 architecture which is a six-instruction issue superscalar machine running at 66.7 MHz. The six instructions consist of two branch-related instructions, two fixed-point instructions, and two floating-point instructions. There are two types of nodes: thin nodes and wide nodes. The main differences between thin and wide nodes are their memory and disk capacities. Thin nodes have up to 512 MB of main memory, 4 GB disk, and four micro channels. Wide nodes come with up to 2 GB of main memory, 8 GB disk, and eight micro channels. The processors used in the experiments reported in this paper are all based on wide nodes. The nodes are connected through a high performance switch, called the Vulcan chip. Each chip connects up to eight processors. Eight Vulcan switching chips comprise a switching board. Figure 1 shows one switch board where eight Vulcan chips connect 32 processors. Figure 2 shows the SP-2 communication adaptor. The adaptor consisting of a 4KB FIFO and an Intel i860 processor that is designed to take the burden of communication-related activities off the main processor.

By introducing a communication co-processor, the main processor can perform computations while the communication processor handles the task of sending/receiving messages to/from other processors. The bidirectional FIFO highlighted in Fig. 2 is central to overlapping communication and computation. The buffer can hold up to 4KB of data

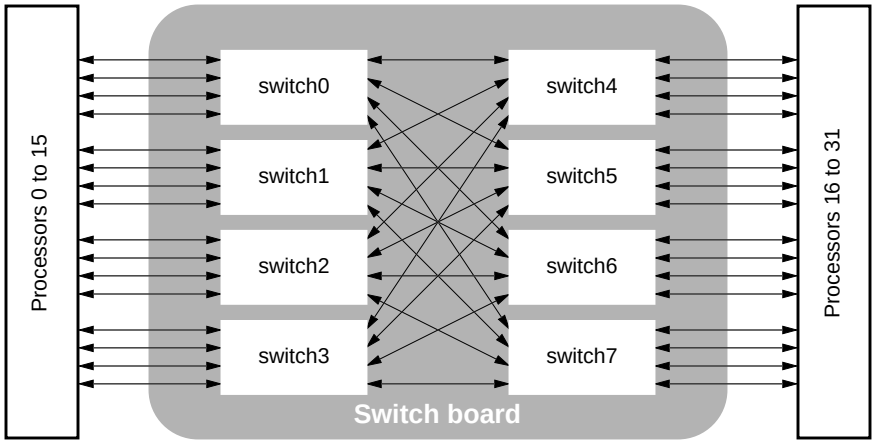

Figure 1: An SP-2 with 32 processors connected through a switch board of eight Vulcan switches.

for two directions or 2KB for each direction. The i860 initiates the sending/receiving of messages. As we shall demonstrate later, this 4KB of FIFO plays a central role in providing the capability of overlapping computation with communication. Other components in the adaptor help realize this overlapping.

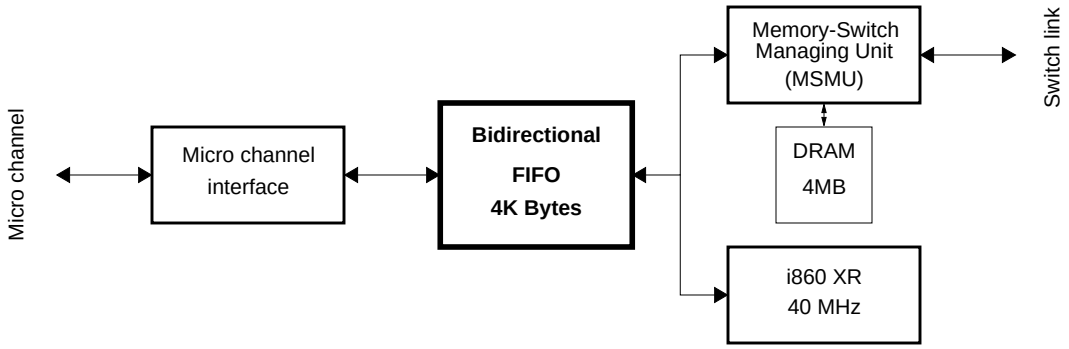

Figure 2: The SP-2 communication adaptor for overlapping communication and computation.

2.2 SGI Power CHALLENGEarray

The SGI Power CHALLENGEarray (PCArray) located at NASA Ames Research Center is a cluster containing four Power Challenge nodes. There are a few other nodes, but they were not used for this work. Figure 3 shows the PCArray consisting of four nodes. Each node is a shared-memory single-address space multiprocessor with 2 GB of main memory. Four nodes are connected through a 16x16 HiPPI switch. Each node has two HiPPI connections. Data transfer between nodes are done with a special HiPPI driver that bypasses, but coexists with, IP traffic. HiPPI transfers can achieve about 110 μs latency and 92 MB/s bandwidth. Very large data transfers are striped across multiple HiPPI connections, giving about 160 MB/s over two HiPPI connections.

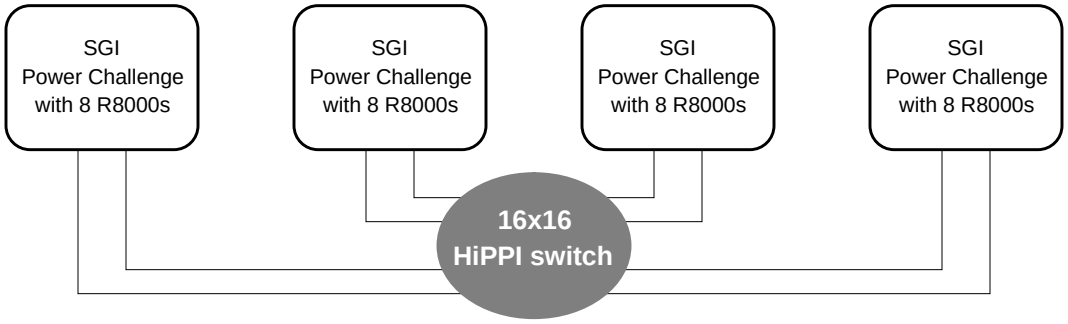

Figure 3: A Power CHALLENGEarray of four nodes, each of which has eight MIPS R8000 processors.

Each nodes consists of eight MIPS R8000 processors running at 90 MHz with 4-issue superscalar capability. Each processor has a peak floating point performance of 360 Mflops. A SGI optimized version of Message-Passing Interface (MPI) is provided to handle communication between nodes. Within a single node, MPI transfers are done through a single-copy mechanism, achieving 64 MB/s bandwidth and 18 μs latency. There are other connections for communication between nodes and for file I/O including ATM, FDDI, Ethernet, SCSI, etc. However those connections are not shown in Fig. 3 as they are not relevant for the work reported in the paper. The major difference between the PCArray and the SP-2 is that the PCArray does not provide any special hardware for fast communication while the SP-2 has an i860 communication co-processor. Hence, there is little possibility of overlapping communication with computation for the PCArray.

3 Problem Descriptions

Two benchmark problems, bitonic sorting and FFT, are described in the context of overlapping computation and communication. Bitonic sorting has moderate parallelism while FFT is highly parallel. To improve the performance of bitonic sorting, we introduce a new communication-efficient algorithm that utilizes overlapping.

3.1 Overlapped Bitonic Sorting

Bitonic sorting introduced by Batcher [4] consists of two steps: *local sort* and *merge*. Figure 4 illustrates bitonic sorting of 32 elements on 8 processors, i.e., n=32 and P=8. For example, consider processors 0 and 1 of bitonic generation with i=0, j=0. P0 has local list (5,13,24,32) and P1 has local list (6,14,23,31) after the local sorting step. P0 and P1 will sort eight elements in an ascending order as indicated by shaded circles. Hollow circles indicate processors that sort elements in a descending order. A line between two processors indicates communication. P0 sends its local list to its mate processor P1 while P1 sends its local list to its mate P0. Both P0 and P1 then merge the list received from their respective mates with their local list. Since P0 has a smaller pid than P1, it takes the lower half (5,6,13,14) while P1 takes the higher half (23,24,31,32). This type of sending, receiving, and merging operations continues until the 32 elements are sorted across the eight processors.

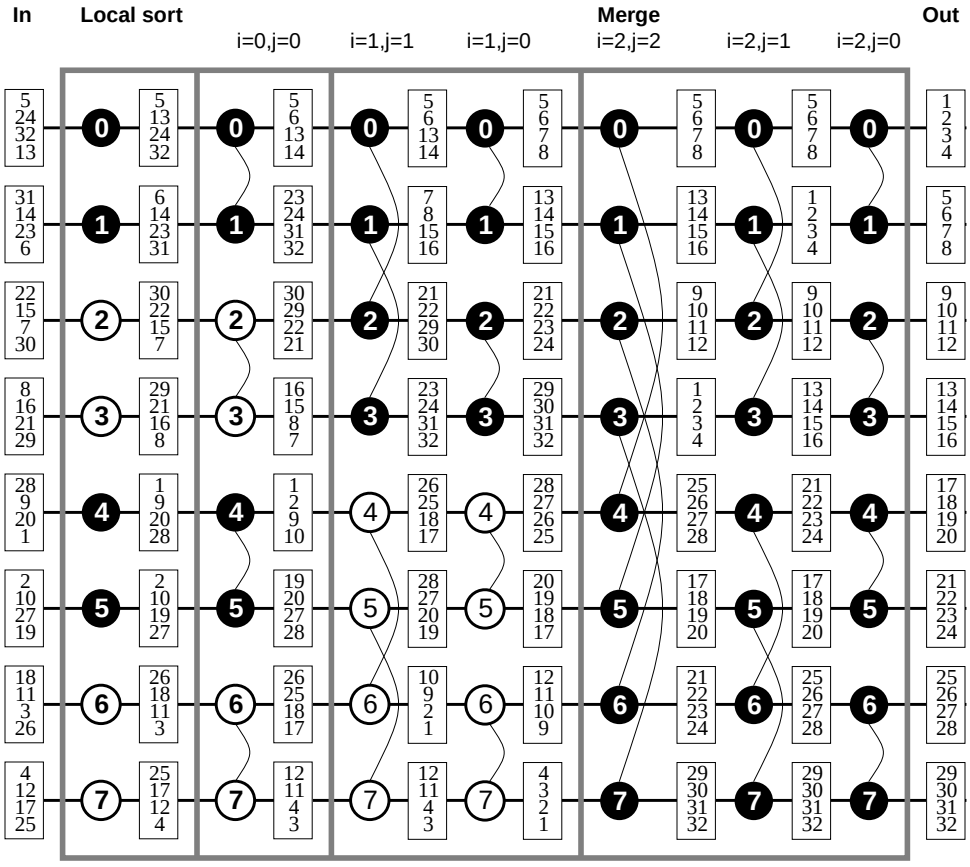

Figure 4: Bitonic sorting of 32 elements on eight processors. Shaded circles indicate those processors performing ascending order merge while hollow circles indicate processors performing descending order merge. Lines connecting two processors indicate communication to send/receive data.

An overlapped version of bitonic sorting divides a list into *s* segments, where each segment holds *m*=*n*/*sP* elements. To handle these segments, a *k*-loop is introduced. Each iteration of the *k*-loop is responsible for merging *m* integers. The pseudo code in Figure 5 describes our overlapped version of bitonic sorting. The main idea of the overlapped version is to first *post* a reading of a segment for the *next* iteration while computation takes place for the *current* iteration. Lines 5, 6, 9, and 10 are key to overlapping. Lines 5 and 6 post sending/receiving a segment of *m* elements for the very *first* iteration of the *k*-loop. The postings are non-blocking operations which return as soon as the posting is done. Upon entering the *k*-loop, another pair of send and receive commands for *m* elements is posted for the next iteration, i.e., *k*=1, while the sending and receiving for *k*=0 is still outstanding. At this moment, two receives are pending for iterations *k*=0 and 1. The *k*-loop then reaches the checking stage in Line 12 and waits until the receive for the *k*=0 iteration is completed. Until this point, there is no overlapping of computation with communication.

```
 1  Local sort L1;                                          /* L1 is my list, L2 is the mate's list, L1=L2=n/P elements */
 2  for (i=0; i<log P; i++) {                               /* P = number of processors */
 3        for (j=i; j<=0; j++)
 4              mpid = find_mate_pid(pid);                  /* mpid = mate processor number, pid = my processor number */
 5              Post non-blocking send of segment 0 of L1   /* function returns after posting */
 6              Post non-blocking recv of segment 0 of L1
 7              for (k=0;k<s;k++) {                         /* n/P elements are split into s segments (messages) */
 8                    if (k < s–1) {                        /* nonblocking send/recv for the next k-th iteration */
 9                          Post non-blocking send of segment k+1 of L1
10                          Post non-blocking recv of segment k+1 of L1
11                    }
12                    Wait until segment k of L2 arrives from mpid
13                    Merge_sort segment k of L2 and L1 to L /* L=2n/P, a buffer to hold the merged result of L1 and L2 */
14              }
15              Take appropriate half of L depending on the values of pid and mpid
16        }
17  }
```

Figure 5: Pseudo code for overlapped bitonic sorting.

As soon as the waiting step of Line 12 returns, the computation for iteration *k*=0 commences by merging segment 0 of L2 into its own list L1, as shown in Line 13. Overlapping occurs at this time because computations for *k*=0 take place while the receive for *k*=1 is either outstanding or taking place within a communication co-processor/bypassing mechanism. By the time the merging is complete for *k*=0, it is expected that the communication for *k*=1 is completed or near completion. This receiving (communication) for iteration *k*+1 and merging (computation) for iteration *k* take place simultaneously, resulting in the overlap of computation with communication. If the computation time were larger than the communication time, it would be possible to complete the communication for the next iteration. The ratio of computation time to communication time is, therefore, one of the key parameters to determine the efficiency of overlapping. The find_mate_pid(pid) function in Line 4 is non-trivial. It computes the order in which segments are sent and received and the order in which they are to be merged. However, the algorithm used is beyond the scope of this paper and is irrelevant for our current objective of investigating communication capabilities of parallel machines.

3.2 Overlapped Fast Fourier Transform

The second problem used in this study is the Fast Fourier Transform (FFT). Figure 6 shows an implementation of FFT with 16 elements on four processors. The 16 data elements are divided into four groups, each of which is then assigned

to a processor. Processor 0, or P0, has elements 0 to 3, P1 has elements 4 to 7, and so on. A FFT with n elements requires $\log n$ iterations. The butterfly shown in Figure 6 thus requires four iterations. In the first iteration, each processor obtains a copy of the four elements assigned to its mate processor. P0 finds P2 as its mate processor, from which elements 8 to 11 are received. Similarly, P1 obtains elements 12 to 15 from P3. P2 and P3 also obtain the data allocated to P0 and P1, respectively. The second iteration is essentially the same as the first iteration, except that the logical communication distance is halved. So, P0 remote receives from P1 this time the four elements which have been *newly* computed by P1 in iteration 0. Similarly, P1 obtains the newly computed four elements from P0. P2 and P3 perform similar operations. The last two iterations do not need communication since the required data are stored locally on the processors. In general, an FFT with blocked data distribution of n elements on P processors requires communication only for the first $\log P$ iterations. The remaining $(\log n) - (\log P)$ iterations can be performed locally; thus, communication is not necessary.

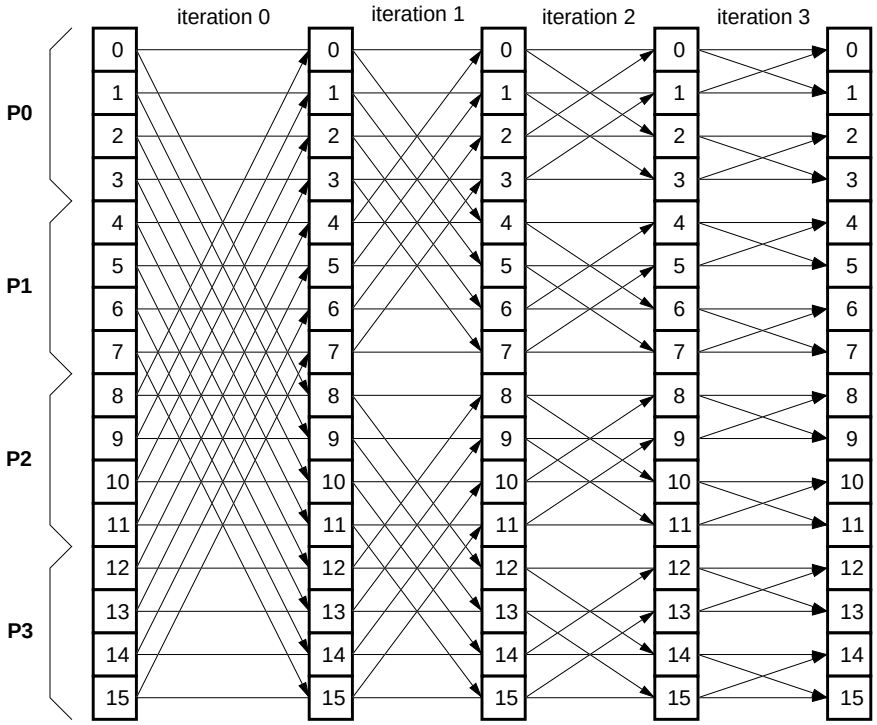

Figure 6: FFT with 16 elements and four processors. Each processor is assigned four elements using blocked data distribution. The first two iterations require communication while the remaining are local computations.

Overlapping computation with communication for FFT is straightforward. Like bitonic sorting, the data assigned to each processor is grouped into segments to control the granularity of communication. Unlike bitonic sorting, however, FFT possesses no data dependence between elements within an iteration. This feature allows computation to commence whenever *any* segment is received from the mate processor. The pseudo-code in Figure 7 illustrates how overlapping is realized. Note that FFT needs $\log n$ iterations but the loop is for only the $\log P$ iterations that require communication.

Within the i-loop is a j-loop which sends/receives s messages (or segments). The size m of a segment is determined as n/sP. This j-loop is not necessary if the entire data of n/P elements is sent/received as a single message. Line 2 finds a mate processor to send/receive data. Lines 3, 4, 7, and 8 are key to overlapping. Lines 3 and 4 post sending/receiving

commands of *m* elements for the very *first* iteration of the *i*-loop. Note that FFT requires two sends instead of one in sorting. This is because every data element has a real and an imaginary part. Receiving is similar, posting two receives for the real and imaginary parts. Upon entering the *j*-loop, another pair of sends and receives is issued for the next *j* iteration, i.e., *j*=1, while sending and receiving for *j*=0 are still outstanding. At this moment, two pairs of sends and receives are pending for *j*=0 and 1. The loop then waits (Line 10) until the two receives posted earlier completes. The rest is similar to bitonic sorting, except that the computation is different.

```
1   for (i=0; i<log P; i++) {                          /* P = number of processors */
2       mpid = find_mate_pid(pid);                     /* mpid = mate processor number, pid = my processor number */
3       Post two non-blocking sends of real and imaginary segment 0/* function returns after posting */
4       Post two non-blocking recvs of real and imaginary segment 0
5       for (j=0;j<s;j++) {                            /* n/P elements are split into s segments (messages) */
6           if (j < s–1) {                             /* nonblocking send/recv for the next j-th iteration */
7               Post two non-blocking sends of real and imaginary segment j+1
8               Post two non-blocking recvs of real and imaginary segment j+1
9           }
10          Wait until real and imaginary segment j arrives from mpid
11          /* Perform computation with segment j */
12      }
13  }
```

Figure 7: Pseudo code for overlapped FFT.

4 Experiment Settings and Overall Results

The two benchmark problems, fine-grain bitonic sorting and FFT, have been implemented on two machines: an IBM SP-2 and a SGI Power CHALLENGEarray. Both algorithms are written in C with MPI [15]. Since their implementations do not require a special environment, they are portable and can be run on any machine which supports MPI. To measure the effectiveness of the overlapping capability, loops were forced to execute synchronously by inserting a *barrier* at the end of the *j*-loop for bitonic sorting (cf. Fig. 5) and the *i*-loop for FFT (cf. Fig. 7). While the problems do not need to run synchronously, we did it only to obtain individual timings. The overall execution times will improve if the barriers were removed. All other communication activities are done in a non-blocking fashion.

Measuring communication time is complicated. We timed MPI_Wait() and the non-blocking MPI_Isend() and MPI_Irecv(). A large portion of the communication time comes from MPI_Wait(). The communication times include remote memory latency, overhead for message preparation, and barrier synchronization times. Computation times are measured indirectly by subtracting the communication time from the total time. It is likely that the computation times include some interrupt times for message handling. Results presented in later sections show however that the computation times are fairly stable, indicating that the interrupt times are not a significant factor. All timings are the minimum of several runs.

The terms elements and integers are used interchangeably throughout this paper, as are segments and messages. The unit for sorting is *integers* while that for FFT is *points*. An integer is 32 bits. A point consists of real and imaginary parts, each of which is 32 bits. The following is a list of parameters that were used in this study:

- P = number of processors, up to 64 for SP2 and 32 (4 nodes) for PCArray
- n = total number of data elements, up to 64M
- s = number of segments per processor
- $m = n/sP$ = message size, of number of data elements per segment

Communication times for the two test problems for P=8 and 32 and a range of data and segment sizes are plotted in Figs. 8 and 9. The computation times are not included as they are essentially independent of the segment size. The computation times for the two machines are not directly compared since they can be misleading due to different clocks and superscalar capabilities. We use them only to show the ratio of computation time to communication time.

Several key observations can be made from the results shown in Figs. 8 and 9:

- The PCArray does not appear to favor any particular message size m. The SP-2, on the other hand, clearly shows that it favors a range of message sizes when the communication time becomes a minimum.
- The PCArray gives very low flat curves when P=8 because the processors all reside on the same node. Communication time is therefore extremely small.
- Increasing the number of processors has a bigger impact on the communication times for the PCArray than for the SP-2. In particular, the PCArray showed a big increase when using two nodes (P=16) instead of one (P=8).
- Communication times increase linearly with increasing data size n.
- Sorting requires a greater communication time than FFT. Recall that FFT has to send twice as many messages because each element has real and imaginary parts. Hence, the communication times in Fig. 9 should be halved to make a fair comparison with those for sorting in Fig. 8.

It is important to note that the above plots are consistent across machines and problems with different numbers of processors and data sizes. In the following sections, we examine these results in more detail and attempt to explain the reasons behind these observations.

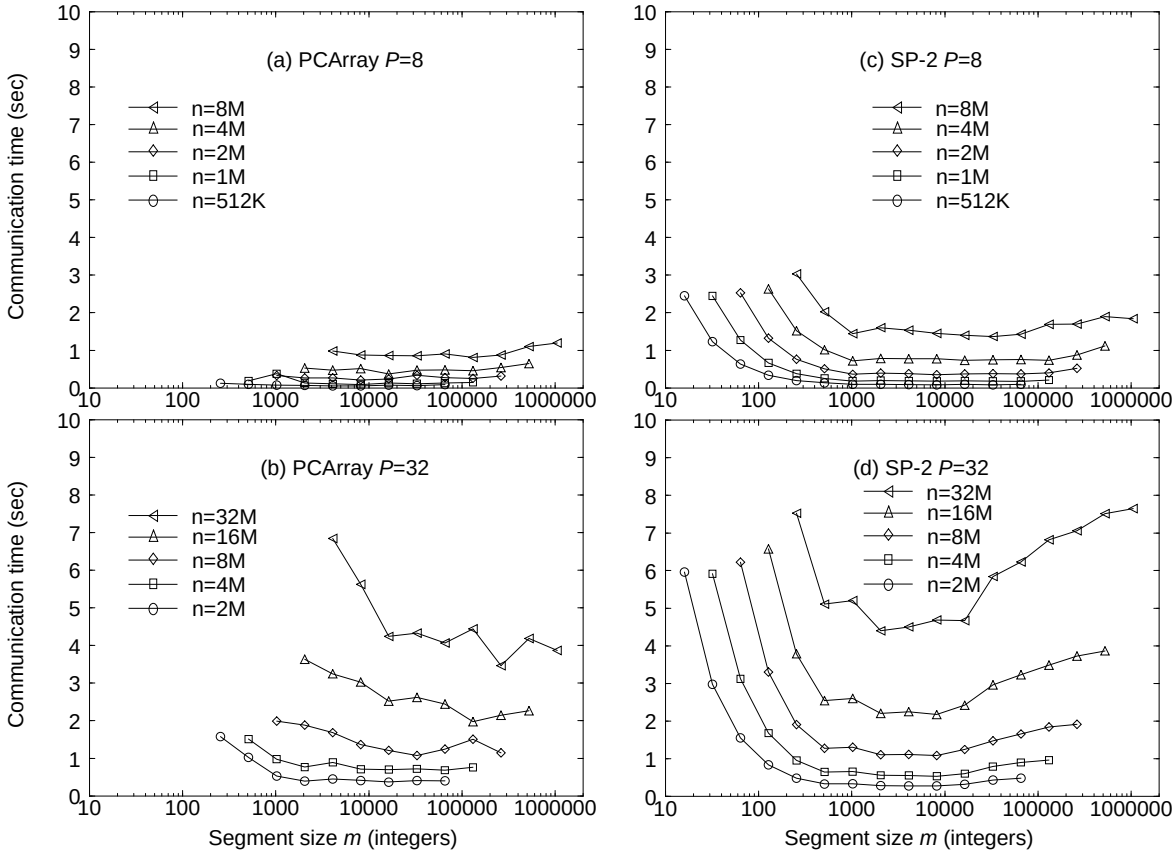

Figure 8: Communication times for fine-grain bitonic sorting.

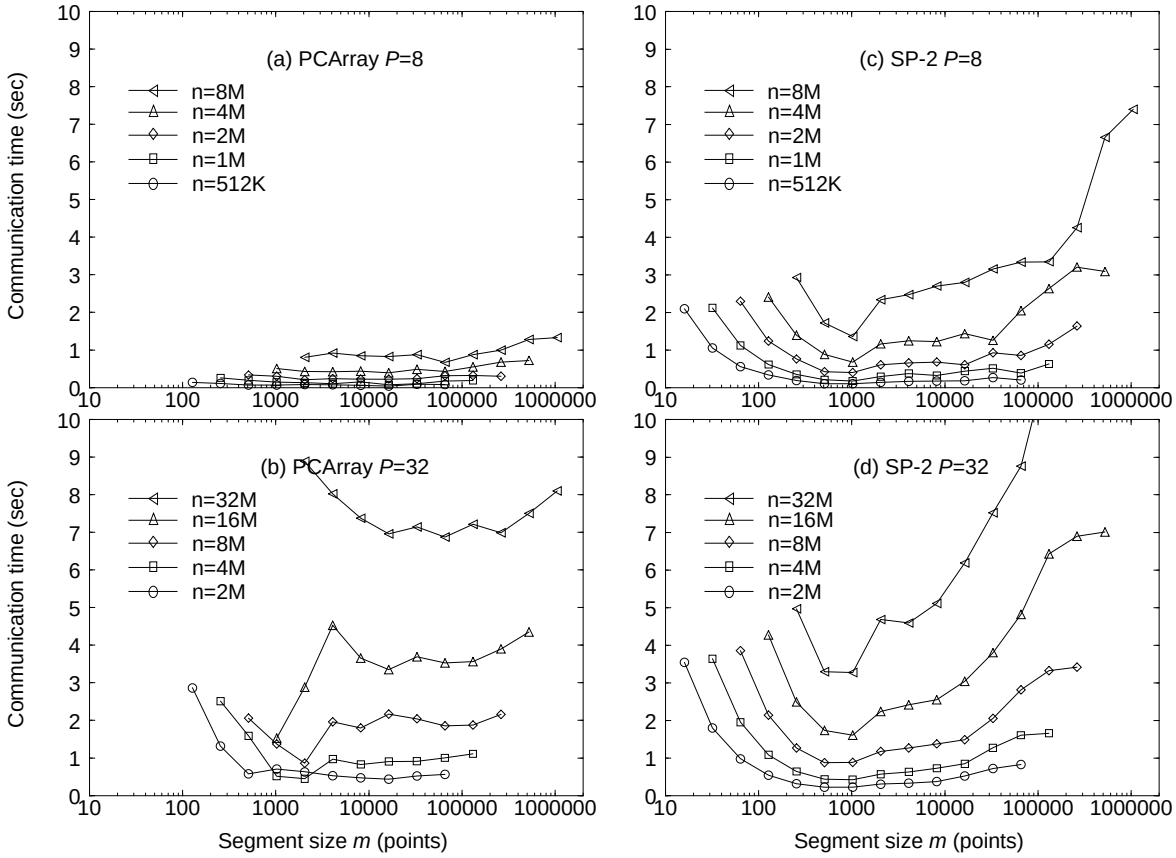

Figure 9: Communication times for FFT.

5 Effects of Message Size across Different Platforms

The plots in Figs. 8 and 9 showed that the two machines gave different performance in terms of communication times for the two problems. The SP-2 showed a relatively consistent behavior while the PCArray gave some fluctuations. Table 1 lists the computation and communication times of the two machines for bitonic sorting with P=32, n=2M and 32M, and s ranging from 1 to 4K.

# of segments s	Sorting: P=32, n=2M					Sorting: P=32, n=32M				
	segment size m	PCArray		SP2		segment size m	PCArray		SP2	
		Comp	Comm	Comp	Comm		Comp	Comm	Comp	Comm
4096	16	*	*	1.248	5.962	256	*	*	4.914	7.524
2048	32	*	*	0.743	2.979	512	*	*	4.409	5.108
1024	64	*	*	0.496	1.555	1K	*	*	4.154	5.202
512	128	*	*	0.371	0.837	2K	*	*	4.018	4.399
256	256	0.523	1.575	0.309	0.483	4K	9.004	6.843	3.944	4.505
128	512	0.496	1.024	0.276	0.331	8K	9.206	5.622	3.917	4.688
64	1K	0.519	0.531	0.261	0.335	16K	9.118	4.240	3.901	4.671
32	2K	0.529	0.398	0.254	0.284	32K	9.232	4.330	3.895	5.843
16	4K	0.524	0.449	0.251	0.277	64K	9.074	4.066	3.883	6.229
8	8K	0.492	0.413	0.247	0.279	128K	9.082	4.440	3.884	6.817
4	16K	0.515	0.370	0.244	0.323	256K	9.415	3.466	3.882	7.057
2	32K	0.495	0.409	0.246	0.435	512K	9.190	4.187	3.887	7.513
1	64K	0.508	0.404	0.244	0.481	1M	9.163	3.868	3.881	7.647

Table 1: Bitonic sorting with P=32. Entries with an asterisk indicate unable to obtain.

The PCArray does not show a preference for any particular message size *m* while the SP-2 clearly favors certain message sizes. It is quite surprising that the PCArray shows little fluctuation in communication time. It should be noted that Table 1 does not list PCArray times when the number of segments *s* is larger than 256. For very small-sized message, we were unable to run the problem instances on the PCArray. The reason we believe is due to the excessive number of segments sent and received simultaneously. This is true for all problem sizes. This rather large number of messages is probably clogging the message handling and the HiPPI interconnection.

If the PCArray communication times were extrapolated for the cases when we had several short messages and were unable to execute, we believe that they would be worse than those for the SP2. The reasons are clear. First, each node of the PCArray is an SMP consisting of eight processors. Within a node, these processors compete for the same memory to send out messages. For the SP-2, each processor is autonomous with its own memory. Second, the PCArray does not provide any hardware support for overlapping message handling and computation. The main processor handles both computation and communication.

Computation times are relatively consistent for both machines except for a few instances on the SP-2 when the message size is small. This is because the computation time includes the interrupt time for message handling. Some overhead are not often properly measured due to the excessive number of small messages. For message sizes over 256 elements, the computation times are highly consistent, indicating that there is enough time to process the messages.

The computation times for the PCArray is consistently twice those for the SP-2 even though the PCArray has a faster clock. This is likely due to the shared-memory nature where processors in each node compete for memory access. Another contributing factor could be that R8000 is a four-issue superscalar processor while Power2 is a six-issue superscalar processor. Table 2 compares the two machines for the FFT problem. Again, the observations that we made for bitonic sorting essentially hold. A more in-depth comparison of the two problems is presented in the next section.

# of segments *s*	FFT: *P*=32, *n*=2M					FFT: *P*=32, *n*=32M				
	segment size *m*	PCArray		SP2		segment size *m*	PCArray		SP2	
		Comp	Comm	Comp	Comm		Comp	Comm	Comp	Comm
4096	16	*	*	1.033	3.549	256	*	*	13.640	4.973
2048	32	*	*	0.897	1.804	512	*	*	13.486	3.300
1024	64	*	*	0.823	0.984	1K	*	*	13.568	3.280
512	128	0.774	2.865	0.796	0.550	2K	14.524	8.886	13.405	4.685
256	256	0.810	1.315	0.775	0.323	4K	14.340	8.023	13.523	4.592
128	512	0.786	0.581	0.761	0.228	8K	14.682	7.372	13.514	5.112
64	1K	0.809	0.710	0.759	0.229	16K	14.901	6.958	13.457	6.191
32	2K	0.795	0.633	0.751	0.311	32K	14.933	7.143	13.315	7.520
16	4K	0.796	0.533	0.757	0.338	64K	14.973	6.870	13.351	8.762
8	8K	0.860	0.476	0.758	0.377	128K	15.080	7.210	13.570	11.709
4	16K	0.894	0.438	0.747	0.527	256K	15.179	6.985	13.323	14.340
2	32K	0.817	0.520	0.774	0.729	512K	15.174	7.505	13.538	14.146
1	64K	0.806	0.563	0.750	0.835	1M	14.994	8.099	13.557	14.123

Table 2: FFT with *P*=32. Entries with an asterisk indicate unable to obtain.

6 Communication Efficiency across Problems

When the two problems are cross-compared, bitonic sorting has a higher communication time than FFT. Figure 10 shows the performance of sorting and FFT on 64 processors of the SP-2 for *n*=4M and 64M. Plots show that sorting requires more communication when the message size is small while FFT has higher communication when the message

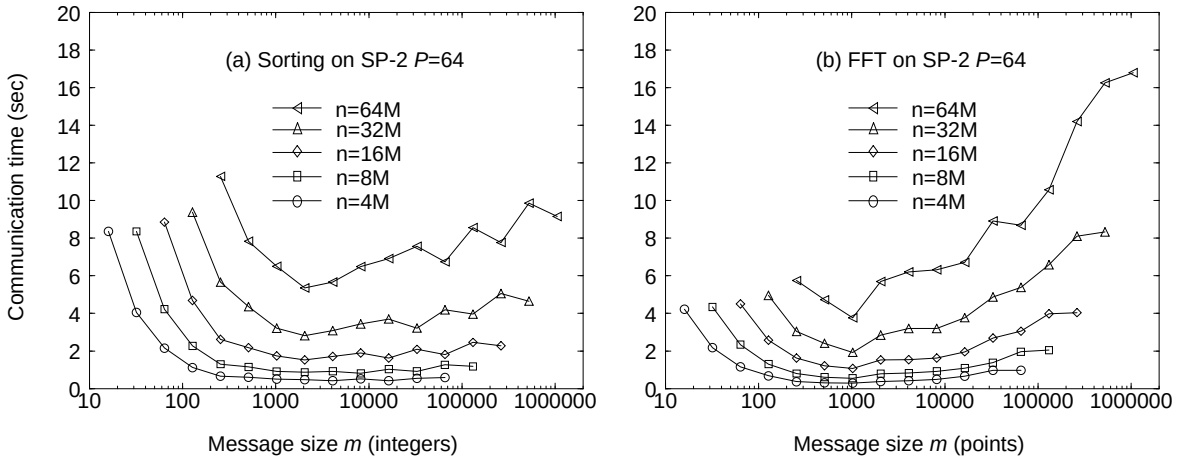

Figure 10: Effects of message size on communication times across problems.

size is large. The main reason is that the amount of messages sent and received is different for the two problems. For sorting, each processor sends a message of size m in each iteration. However, FFT sends twice the amount, i.e., $2m$ elements in each iteration per processor because each data point has real and imaginary parts. Another reason the valley for FFT is shifted to the right is the ratio of computation time to communication time. Table 3 shows some of the computation and communication times for the two problems on the SP-2.

It is clear from Table 3 that the ratio of computation to communication is different for the two problems. The ratio for sorting is below one whereas it is generally greater than one for FFT. This difference in ratio suggests that FFT has more time to process the messages. The impact of communication for FFT is smaller compared to that for sorting, thereby resulting in a higher overall performance. More details are presented in the following section.

# of segments s	SP-2: P=64, n=4M							SP-2: P=64, n=64M						
	segment size m	Sorting			FFT			segment size m	Sorting			FFT		
		Comp	Comm	Ratio	Comp	Comm	Ratio		Comp	Comm	Ratio	Comp	Comm	Ratio
4096	16	0.565	8.356	0.068	1.310	4.222	0.310	256	6.125	11.285	0.543	17.047	5.744	2.968
2048	32	0.466	4.053	0.115	1.111	2.177	0.510	512	6.095	7.834	0.778	16.683	4.724	3.532
1024	64	0.431	2.159	0.200	1.025	1.163	0.881	1K	6.019	6.504	0.925	16.834	3.761	4.476
512	128	0.402	1.124	0.358	0.973	0.685	1.420	2K	5.985	5.356	1.117	16.823	5.691	2.956
256	256	0.394	0.666	0.592	0.977	0.373	2.619	4K	5.979	5.671	1.054	16.493	6.201	2.660
128	512	0.386	0.603	0.640	0.959	0.305	3.144	8K	5.970	6.488	0.920	16.599	6.312	2.630
64	1K	0.384	0.511	0.751	0.937	0.294	3.187	16K	5.997	6.910	0.868	16.767	6.708	2.500
32	2K	0.380	0.472	0.805	0.961	0.376	2.556	32K	6.005	7.561	0.794	16.548	8.912	1.857
16	4K	0.379	0.418	0.907	0.953	0.424	2.248	64K	5.961	6.737	0.885	16.814	8.684	1.936
8	8K	0.377	0.524	0.719	0.947	0.494	1.917	128K	5.968	8.563	0.697	16.602	10.572	1.570
4	16K	0.377	0.423	0.891	0.939	0.666	1.410	256K	5.974	7.771	0.769	16.608	14.195	1.170
2	32K	0.378	0.556	0.680	0.951	0.978	0.972	512K	5.990	9.863	0.607	16.813	16.247	1.035
1	64K	0.376	0.592	0.635	0.934	0.981	0.952	1M	5.968	9.157	0.652	16.591	16.787	0.988

Table 3: Comparison between sorting and FFT on the SP-2 with P=64.

7 Efficiency of Overlapping

Overlapping communication with computation is central to obtaining high performance on parallel machines. Summarizing our findings in this study, we identify the efficiency of overlapping of the two machines. To measure the overlapping efficiency, it is important to accurately define the basis on which the communication times are compared.

When only one segment is used, i.e., s=1, it is not possible to overlap computation with communication since computation can begin only after the communication of the entire segment is complete. Let $T_{comm,s}$ be the communication time for s segments. We define the efficiency of overlapping as $E = (T_{comm,1} - T_{comm,s}) / T_{comm,1}$. Figure 11 identifies the capabilities of overlapping for the two machines for sorting. The SP-2 shows a much higher overlapping than the PCArray.

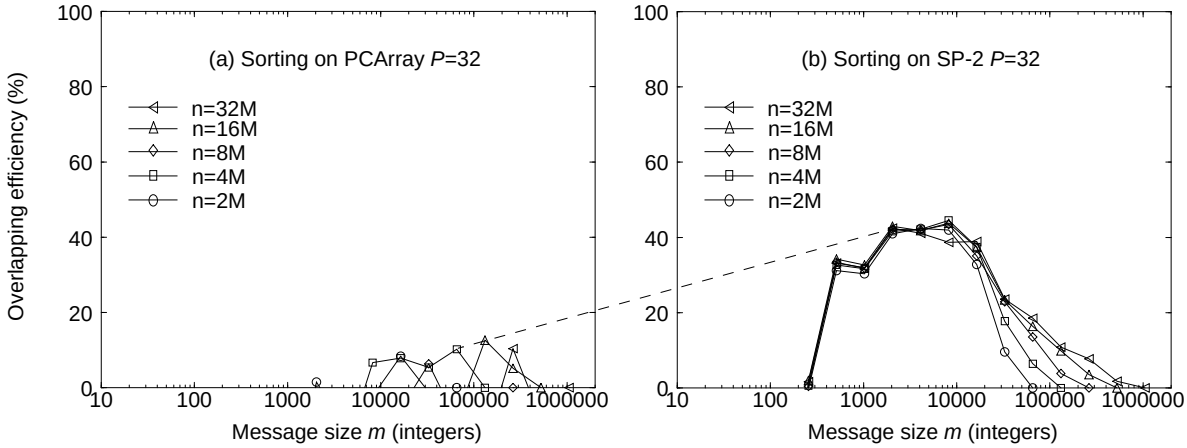

Figure 11: Efficiency of overlapping for bitonic sorting with P=32.

Figure 11 indicates that the PCArray exhibits an overlapping efficiency of merely 10%. The SP-2, however, shows an efficiency of over 40% for message sizes between 2K and 16K integers. The PCArray shows little overlapping because the machine does not provide any hardware that can handle messages independent of the main processor R8000. Every message sent and received uses the main processor cycles. The SP-2, on the other hand, provides two hardware support for handling communication: the i860 communication co-processor and the 4KB bidirectional buffer. Figure 11(b) clearly indicates that the buffer is effectively utilized when the message size is about 4KB.

When the message size is too small or too large, the buffer is no longer effective since other factors such as overhead and barrier synchronization time then begin to become important. When the message size is too small, the overhead to execute instructions for message preparation becomes the dominant factor. This overhead is a fixed cost and cannot be overlapped with computation since the instructions must be executed to prepare messages, regardless of the message size. When the message size is too large, the overlapping efficiency is drastically reduced since the 4KB message buffer is too small. While there is certainly very little overhead, this large-sized messages will occupy the communication channels longer to complete sending/receiving each message, resulting in clogging the bandwidth.

The ratio of computation to communication is also a key factor in determining the overlapping efficiency. If hardware support is provided and communication takes place while computation proceeds, the computation time must be larger than the communication time to effectively mask the communication. To compare the efficiency in terms of the ratio of computation to communication, we have plotted the overlapping efficiency for FFT in Fig. 12. We find that the efficiency of the SP-2 has increased to 80% while that for the PCArray is still low, approximately 20%. Recall from Table 3 that the ratio of computation to communication times for the FFT is generally more than twice that for sorting on both machines. This is directly reflected in the overlapping efficiency of FFT compared to sorting.

The overlapping efficiency for the PCArray in Fig. 12(a) shows unexpected behavior for message sizes 1K and 2K. The dotted circle of Figure 12(a) show a very high overlapping efficiency of 60%. We believe these spikes are due

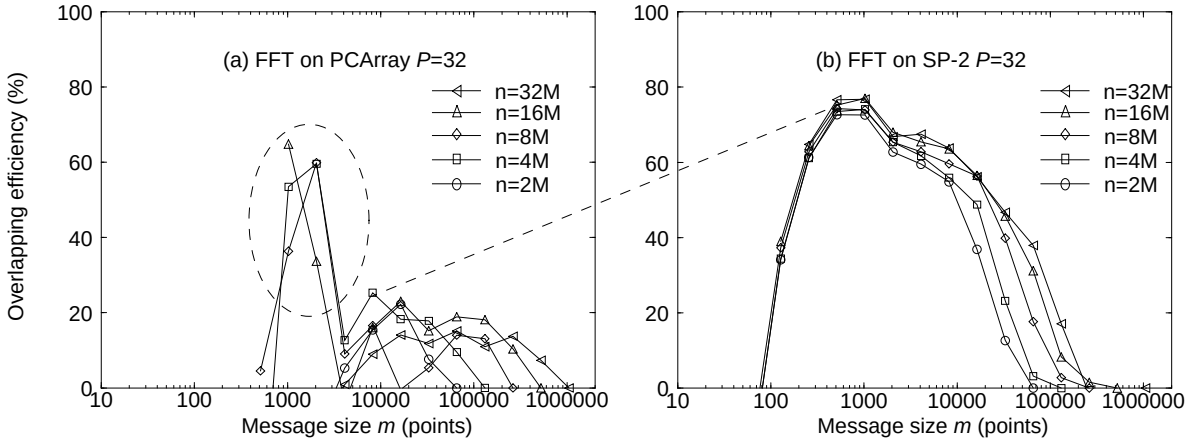

Figure 12: Efficiency of overlapping for FFT with P=32. The dotted circle indicates that there are some errors in measurement possibly due to resident processes running on the machine.

to errors in measurement and do not reflect true performance. Measuring the communication time is indeed not straightforward due to many practical difficulties. The right hand side of Figure 12(a) should be considered the true overlapping efficiency for the PCArray since it is consistent with the results for sorting.

When the number of processors is increased to 64 on the SP-2, the overlapping efficiency remains consistent for both sorting and FFT as depicted in Fig. 13. According to our experimental results, the overlapping efficiency of sorting is always about half that for FFT, regardless of the number of processors. However, both sorting and FFT show little change when the number of processors is increased from 32 to 64. (Compare Figs. 11(b) and 13(a) for sorting, Figs. 12(b) and 13(b) for FFT.) This consistency in overlapping efficiency indicates that the number of processors is not the main factor contributing to communication overlapping.

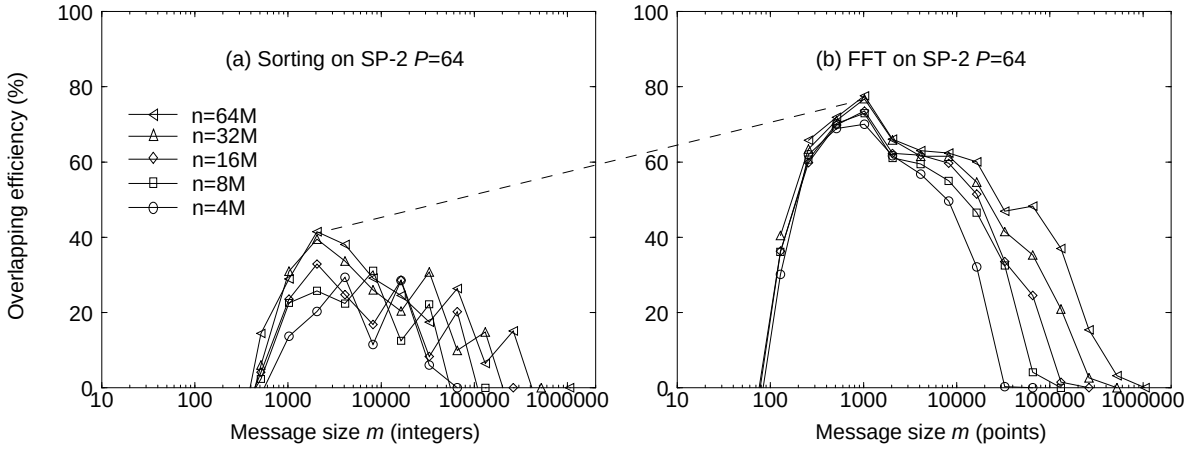

Figure 13: Comparison of overlapping efficiency between sorting and FFT on the SP-2 with P=64.

For large numbers of processors, the synchronization and overhead times become increasingly important in deciding the overlapping efficiency since the synchronization cost increases with the number of processors. Barrier synchronization was used at the end of each iteration to synchronize processors and measure various timings. This synchronization cost is a fixed cost and cannot be overlapped with computation, regardless of the message size. Experimental results indeed indicate that when the number of processors is 64, the synchronization cost alone is more than twice the latency.

8 Conclusions

Programming parallel machines involves numerous practical concerns. This paper has investigated the interplay between problems and communication capabilities of parallel machines. To demonstrate the practicality of our study, we have fixed the parallel programming paradigm to Message Passing Interface. Two machines were selected for experiments: an IBM SP-2 distributed-memory multiprocessor and an SGI PowerCHALLENGEarray (PCArray) symmetric multiprocessor. Two benchmark problems, bitonic sorting and Fast Fourier Transform, have been selected for experiments. Communication-efficient algorithms are developed to exploit the overlapping capabilities of the machines. Programs have been written in MPI for portability and identical codes are used for both machines. Various data sizes and message sizes are used to test the machines' communication capabilities.

Experimental results have indicated that the communication performance of the multiprocessors are consistent with the size of messages. The PCArray is not highly sensitive to message size but yielded low communication overlapping. In fact, it yielded only 20% overlapping of communication with computation for FFT. The main reason is because SGI provides no hardware support for message passing. Our results and analysis are based only on up to 32 processors of the PCArray. It is not clear whether similar performance will be obtained for 64 processors. The SP-2, on the other hand, has shown sensitivity to message size but yielded high overlapping of communication with computation. It clearly favored a certain range of message sizes. An overlapping efficiency of up to 80% was obtained for FFT. This high value can be attributed to the communication co-processor and its 4KB message buffer.

When the two test problems are cross-compared, sorting required a greater communication time than FFT. The reason is because the ratio of computation time to communication time for sorting is much lower than that for FFT. This low computation time for sorting has been found to be insufficient to mask some latency by the communication co-processor. Another reason is that FFT has to send twice as many messages because each element has real and imaginary parts. Note that the FFT communication times shown in the paper should be halved for each message size since they are based on real and imaginary parts.

Increasing the number of processors has a bigger impact on the communication times for the PCArray than for the SP-2. In particular, the PCArray showed a big increase when using two nodes (16 processors) instead of one (8 processors). This is due to the fact the one node does not involve any communication since it is a shared-memory machine by itself. However, when two nodes are used, the communication time drastically increased because of communication through HiPPI switches. The communication behavior of the SP-2 has been shown to be consistent as the number of processors was increased to 64. We found that the overhead and synchronization times become significant which limit the overlapping efficiency for large number of processors. We are currently working on Cray T3E to further compare the communication performance with the SP-2 and the PCArray.

References

1. T. Agerwala, J. L. Martin, J. H. Mirza, D. C. Sadler, D. M. Dias, and M. Snir, "SP-2 System Architecture", in *IBM Systems Journal* Vol. 34, No. 2, 1995
2. A. Alexandrov, M. Ionescu, K. Schauser, and C. Scheiman, "LogGP: Incorporating Long Messages into the LogP Model - One step closer towards a realistic model for parallel computation," in *Proc. of the 7th ACM Symposium on Parallel Algorithms and Architectures*, Santa Barbara, CA, July 1995, pp. 95-105.
3. R. Alverson, D. Callahan, D. Cummings, B. Koblenz, A. Porterfield, and B. Smith, "The Tera computer system," In *Proceedings of ACM International Conference on Supercomputing*, Amsterdam, Netherlands, June 1990, ACM, pp.1-6

4. K. Batcher, “Sorting Networks and Their Applications,” in Proc. the AFIPS Spring Joint Computer Conference 32, Reston, VA, 1968, pp.307-314.

5. D. Cann, The Optimizing SISAL Compiler: Version 12.0, Lawrence Livermore National Laboratory, 1992.

6. D. Culler, R.M. Karp, D.A. Patterson, A. Sahay, E.E. Schauser, E. Santos, R. Subramonian, and T. von Eicken, “LogP: Towards a Realistic Model of Parallel Computation,” in Proc. of the 4th ACM Symposium on Principles and Practice of Parallel Programming, San Diego, CA, May 1993.

7. J.T. Feo, Cann, D.C., and Oldehoeft, R.R., “A Report on the SISAL Language Project,” *Journal of Parallel and Distributed Computing* 10, December 1990, pp.349-365.

8. G. Gao, L. Bic and J-L. Gaudiot (Eds.) Advanced Topic in Dataflow Computing and Multithreading, IEEE Computer society press, 1995

9. High Performance Fortran Forum, High Performance Fortran Language Specification version 2.0, Center for Research on Parallel Computation, Rice University, Houston, TX, November 1996.

10. R. Iannucci, G. Gao, R. Halstead, and B. Smith (Eds.), Multithreaded Computer Architecture, Kluwer Publishers, Norwell, MA 1994

11. K. Kennedy and U. Kremer, Automatic Data Layout for High Performance Fortran, In *Proceedings of Supercomputing’95*, San Diego, CA, December 1995.

12. K. Kennedy, N. Nedeljkovic, and A. Sethi, Communication Generation for Cyclic(k) Distributions. Languages, Compilers and Run-Time Systems for Scalable Computers, Kluwer Academic Publishers, May, 1995.

13. Y. Kodama, H. Sakane, M. Sato, H. Yamana, S. Sakai, and Y. Yamaguchi, “The EM-X Parallel Computer: Architecture and Basic Performance,” in *Proceedings of ACM International Symposium on Computer Architecture*, Santa Margherita Ligure, Italy, June 1995, pp.14-23.

14. J. R. McGraw, Skedzielewski, S.K., Allan, S.J., Oldehoeft, R.R., Glauert, J., Kirkham, C., Noyce, W., and Thomas, R., “SISAL: Streams and Iteration in a Single Assignment Language: Reference Manual version 1.2,” *Manual* M-146, Rev. 1, Lawrence Livermore Laboratory, Livermore, CA, 1985.

15. Message Passing Interface Forum, MPI: Message-Passing Interface Standard, Version 1.1, Technical Report, University of Tennessee, Knoxville, TN, June 1995.

16. R. Nikhil, “Id (Version 90.1) Reference Manual,” MIT CSG Memo 284-2, July 1991.

17. R. Nikhil, G. Papadopolous, and Arvind, “*T: A Multithreaded Massively Parallel Architecture,” in *Proceedings of ACM International Symposium on Computer Architecture*, Gold Coast, Australia, May 1992, pp.156-167.

18. G. Papadopolous, An Implementation of General Purpose Dataflow Multiprocessor, MIT Press, Cambridge, MA, 1991.

19. B. J. Smith, “A Pipelined, Shared Resource MIMD Computer,” in *Proceedings of International Conference on Parallel Processing*, 1978, pp.6-8.

20. A. Sohn, M. Sato, N. Yoo, and J-L Gaudiot, Data and Workload Distribution in a Multithreaded Architecture, *Journal of Parallel and Distributed Computing*, December 1996.

21. C. B. Stunkel, D. G. Shea, B. Abali, M. Atkins, C. A. Bender, D. G. Grice, P. H. Hochschild, D. J. Joseph, B. J. Nathanson, R. A. Swetz, R. F. Stucke, M. Tsao, and P. R. Varker, “The SP-2 Communication Subsystem,” Technical Report, IBM T. J. Watson Research Center, August 1994.

22. L. G. Valiant, “A Bridging Model for Parallel Computation,” *Communications of the ACM* 33 (8), August 1990, pp.103-111.